The Processing Graph Method Tool (PGMT)

Richard S. Stevens
Naval Research Laboratory
Washington, DC 20375-5337
stevens@ait.nrl.navy.mil

Abstract

To acquire state-of-the-art hardware at reduced cost, the U.S. Navy is committed to buying Commercial Off The Shelf (COTS) computer hardware. In this rapidly changing technological world, today's hardware will be obsolete tomorrow. The Navy's complex problems often require more computational power than can be delivered by a single serial processor. The solution lies in distributed processing. However, distributed processors tend to have architecture specific languages, requiring an expensive and time-consuming manual rewrite of application software as new technology and new machines become available.

The Processing Graph Method (PGM), developed at the Naval Research Laboratory (NRL) in Washington, DC, is an architecture independent method for specifying application software for distributed architectures. Its model of computation is reconfigurable dynamic data flow: dynamic, because the amount of data consumed and produced by an actor may vary from one firing to another; and reconfigurable, because a graph may be disassembled and reassembled. PGM was implemented on the Navy Standard Signal Processor (AN/UYS-2), and on VAX and Sun workstations. The PGMT project at NRL is developing a tool set that will facilitate the implementation of PGM on a given distributed architecture at relatively low cost. We describe the major features PGM and discuss the PGMT project.

1: Introduction

The relentless demand for increasingly high performance is outpacing the increasing speed of sequential processors, and so we turn to distributed architectures. To take advantage of the power of a given distributed architecture, the subject domain expert must be teamed with a highly skilled programmer who can write application code that is tailored to the architecture. As new, higher performance architectures become available, previously written code requires complete revision. This makes software prohibitively expensive to maintain. To lower the life-cycle cost of software, we must have a high-level, user-friendly, architecture-independent language that expresses in a natural way the inherent parallelism in a user application. To implement this language on new distributed architectures at low cost, we need a tool set to automate much of the development of compilers. To take advantage of the power of a given architecture, this tool set must have the capability to analyze an application, and then partition and distribute it within the architecture. How this partitioning and distribution is done can have an enormous effect on performance.

The Processing Graph Method (PGM) [1] was developed at the Naval Research Laboratory (NRL) to meet the needs of a language for developing applications for distributed architectures. A PGM application encompasses a processing graph and a command program. The computation model of the processing graph is data flow, which has a long history that

began in the 1960s [2]. The command program has the capability, by calling library procedures, to reconfigure the graph. In this way the processing graph may be adapted to rapidly changing processing requirements. PGM was implemented on the Navy Standard Signal Processor (AN/UYS-2) for signal processing applications. PGM was also implemented on the VAX and SUN workstations for use in prototyping applications for the AN/UYS-2.

The PGM Tool (PGMT), under development at NRL, will demonstrate the technology to implement PGM on a given heterogeneous network of processors. For further information, see http://www.ait.nrl.navy.mil/pgmt/pgm2.html. Section 2 gives a brief but friendly description of PGM [1]. Section 3 discusses the PGMT project, its technical problems, and the current status of our work. Section 4 is a concluding summary.

2. The Processing Graph Method

Motivated by the structure of Petri Nets [4], PGM defines a processing graph to comprise *transitions* and *places*. A transition (or actor) processes data; a place stores data enroute between transitions. A transition may *fire* when there is sufficient data in its input places. To fire, a transition reads and consumes data from its input places, and produces the results of computation to its output places. While firing, a transition has no internal memory of its previous firings. The state of a graph is determined by the data stored in its places.

A data flow graph is *determinate* if the output data streams are determined solely by the input data streams and are independent of the order in which the actors are fired [2]. A PGM graph, called a *processing graph*, will be determinate if certain constructs are not used, as we will note below. After discussing the fundamental notion of a family, we describe the features of PGM. An example of an application processing graph is given in [3].

2.1 Family

A *family* is a list of *members* of a common type. Each member may be a common base type, e.g., integer, floating point, or a user-defined type. Recursively, each member in a family may be another family, provided every member has the same depth of recursion and the same base type, but not necessarily the same number of members.

2.2 Place

PGM is strongly typed. Each place has an associated *mode* that identifies the type of the tokens that it stores. PGM specifies two kinds of places: *queues* and *graph variables*.

2.2.1 Queue: A *queue* stores data enroute from transition to transition. Each queue stores its data in a family whose members are called *tokens*. A queue *Q* passes tokens from transition *A* to transition *B* in a first-in-first-out manner. If *Q* is empty, then *B* may not fire until *A* produces sufficiently many tokens to *Q*. Each queue has a *capacity*, which determines the maximum number of tokens. If *Q* is at capacity, then *A* may not fire until *B* consumes one or more tokens from *Q*, thereby putting *Q* sufficiently below capacity. A queue's capacity may be adjusted as needed during run time.

2.2.2 Graph Variable: A *graph variable* is a place that stores a single token. Any number of transitions may access a given graph variable for input and/or output. A transition reading a graph variable gets the most recent token produced to that graph variable. Consuming from a graph variable has no effect; the token remains in the graph variable. When a transition

produces a token to a graph variable, the previous token is replaced by the new one. A processing graph with one or more graph variables may not be determinate.

2.3 Transition

For a *transition* to fire, each input place must have sufficiently many tokens available, and each output place must have sufficient capacity to accept the tokens to be produced. PGM distinguishes between *ordinary transitions* and *special transitions*.

2.3.1 Ordinary Transition: In each firing, an *ordinary transition* reads and consumes exactly one token from each of its input places, and produces exactly one token to each of its output places. Each ordinary transition has a user-specified *transition statement*, which defines the computation to be performed during each firing. The transition statement may call routines, or *primitives*, which may be written in native code to enhance performance by taking advantage of individual processor architecture.

Figure 1 depicts an ordinary transition (the circle), reading a single token from each of three input places and producing a single token to each of two output places.

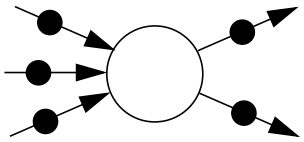

Figure 1: Ordinary Transition

2.3.2 Special Transitions: The *special transitions* do no user-specified processing. Instead, they reformat data. The special transitions support the restructuring of data when it is necessary during one execution to read a number of tokens (zero or more) from an input place or to produce a number of tokens to an output place. The three kinds of special transitions are *pack*, *unpack*, and *uncontrolled merge*.

A *pack* transition is used to read zero or more tokens from a single input queue and assemble them into a single token that is a family of the tokens read in. This is useful, for example, when an ordinary transition reads a token that is a family of numeric values produced as individual tokens by an upstream transition.

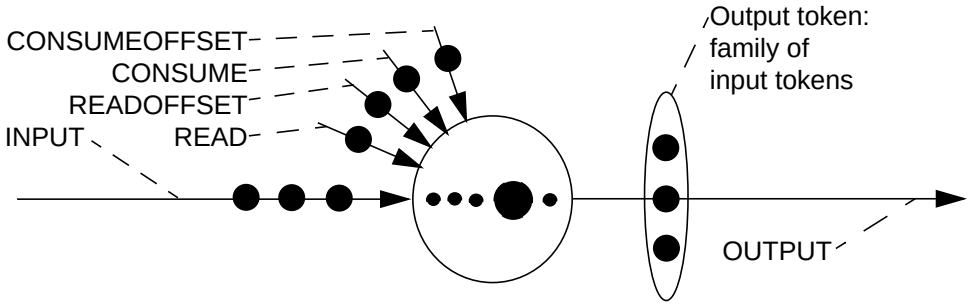

Figure 2: Pack Transition

Each pack transition reads and consumes zero or more tokens from its input place, and produces a single token to its output place. The number of tokens to be read and the number of tokens to be consumed are specified by parameters of the pack transition; these parameters

may be read from other input places. Each pack transition produces a single output token, which is a *family* whose members are the tokens read in. Figure 2 depicts a pack transition, the four input parameters, three input tokens, and one output token.

An *unpack* transition reads a single token that is a family and produces the members as individual tokens. This is useful, for example, when an ordinary transition produces a token that is a family of numeric values that are read as individual tokens by a downstream transition.

Each unpack transition reads and consumes a single token that is a family. This input token is disassembled, and each family member is produced as an output token. The content of the input token is produced as a single integer token to a second output place. Figure 3 depicts an unpack transition with one input token (a family with n members), n output tokens, and the integer n.

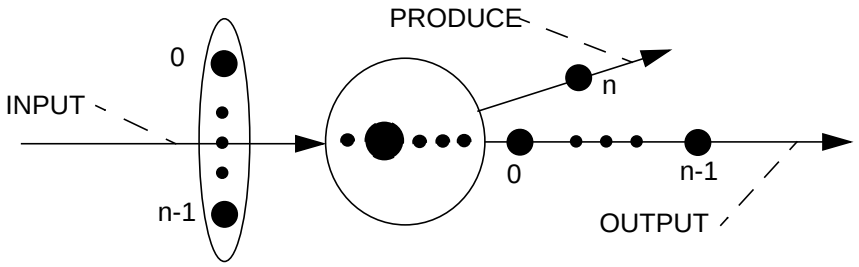

Figure 3: Unpack Transition

Each *uncontrolled merge* transition has a family of input places and a single output place. To fire, it is sufficient for a single token to exist on just one of the input places. Firing consists of reading and consuming one token from one place and producing an equal token to the output place. A processing graph with one or more uncontrolled merge transitions may not be determinate. Figure 4 depicts an uncontrolled merge transition (the circle containing a Venn diagram) with a family of inputs, one token at one input, and one output token. A processing graph with one or more uncontrolled merges may not be determinate.

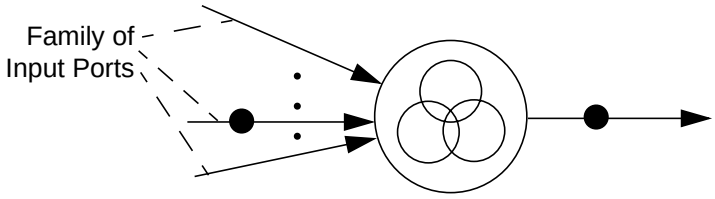

Figure 4: Uncontrolled Merge

2.4 Modular Graph Specification

A processing graph may have one or more *included graphs*. Each included graph may, in turn, have its own included graphs, etc.

PGM applies the notion of family to all PGM objects, including transitions, places, and even included graphs. Thus a user may construct parallel channels of computation in terms of a family of included graphs with a common underlying graph specification.

An example of a processing graph is shown in Figure 5. This graph has no included graphs, however it could be used as an included graph in a larger signal processing application. We use a circle to represent a transition, an open triangle to represent a queue, and a triangle with

line to represent a graph variable. A square represents an included graph, and square with an X represents input ports or output ports of the graph.

The filter and bandshift transitions are ordinary transitions. During each execution, the filter transition reads two tokens, one from each of its inputs. These input tokens are families of the same size; the transition computes their inner product and produces a token containing the result on its output. One of the input tokens contains the filter coefficients. This token is stored in a graph variable with no transition writing to it; thus the graph variable maintains a constant value, and the same filter coefficients are used every time. The other input token to the filter transition is output from the pack transition, which takes its input from the graph's input port at the left. A stream of numbers to be processed by the graph is input at this port.

In the pack transition, we control the size and contents of each output token by setting the READ and CONSUME parameters, with the READOFFSET and CONSUMEOFFSET set to zero. Specifically, the size of its output family token is equal to the READ parameter, which we set to match the number of filter coefficients. We set the CONSUME parameter by an amount less than the READ in order to reuse the values in the input stream.

In each execution of the bandshift transition, the output token is the product of the input token and a factor that changes from one execution to the next. This factor is a function of the previous factor and the center frequency. Because the transition has no memory of previous executions, we provide a feed-back queue to store the previous factor. For the first execution of the bandshift transition, we give this feed-back queue an initial token with value 1. The center frequency, which may be changed, is stored in a graph variable. To allow a new value to be written from an external source, we connect a graph input port to this graph variable.

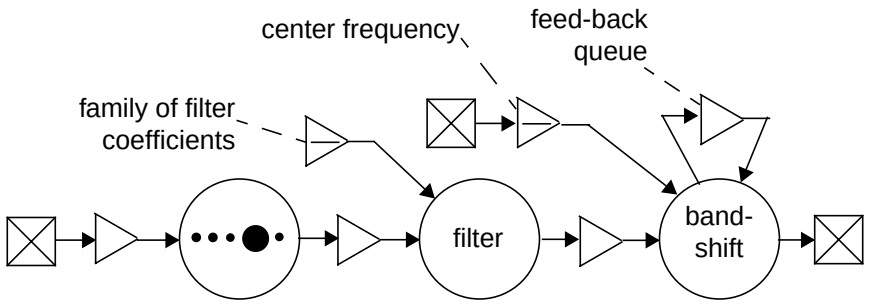

Figure 5: Graph

2.5 Configuration and Reconfiguration: Command Programs

An integral part of PGM is the capability to reconfigure the processing graph. Reconfiguration is specified by a *command program* to restructure the processing graph to meet new processing requirements in a rapidly changing environment.

The user writes the command program in a high-level language like C, C++, Ada, or Java. PGM specifies a library of procedures that the command program may call to do such things as

- create a processing graph and enable its transitions to fire,
- enter data into a processing graph,
- read data output from a processing graph,
- suspend the execution of a processing graph, i.e., disallow transition firing,
- save a processing graph and subsequently reload it,
- modify the values of parameters used by transitions in the processing graph, and
- modify the structure of a processing graph by disconnecting places and transitions and reconnecting them in different ways.

Some of these procedures give the command program the power to alter the values of key parameters that are used in computation. In the above example, the command program may write the new center frequency into that graph variable via the graph input port. Other procedures give greater power to the command program. By disconnecting places, transitions, and included graph ports, and reconnecting them in different ways, the command program may change the processing that is performed in response to dynamically changing requirements. An example is shown in Figure 6.

Data enters this graph via the graph input port at the left and is initially processed by the included graph A. The output from graph A is then further processed by included graph B, the results of which are delivered for external use, e.g., display or storage. If the dynamic situation changes, it may be necessary to process the output of A using included graph C instead of B. In this case, the command program reconfigures the graph by suspending execution of the graph, disconnecting the ports of B (shown by solid arrows), reconnecting the ports of C (shown by dashed arrows), and finally restarting the graph.

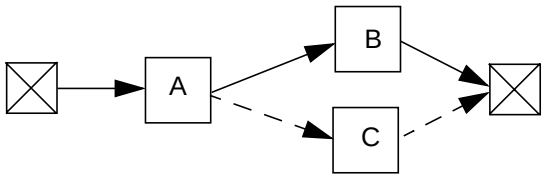

Figure 6: Graph Reconfiguration

3. The Processing Graph Method Tool (PGMT)

NRL is currently developing PGMT to demonstrate a way of implementing PGM applications on a given distributed architecture. A complete implementation of PGM on a target architecture must include a library of the primitives that may be called in transition statements and a library of the command program procedures.

PGMT will provide a tool set that supports the following:

- User specification by Graphic User Interface (GUI) of a target distributed architecture,
- Analysis of the target architecture,
- User specification by GUI of a processing graph,
- Analysis of the processing graph, and
- Partitioning of the processing graph into segments and assignment of the segments to the processors in the target architecture.

The specification and analysis of a given distributed architecture must be performed just once. To analyze the target architecture, the following inputs will be used:

- the number of each kind of processor in the architecture,
- the primitives that can be executed on each kind of processor,
- processing times for each primitive on each kind of processor,
- the communication connections between the processors, and
- communication times between processors.

In most distributed architectures, the time to transmit data between processes is significant when compared to the execution time of transitions. Assigning transitions to processors without concern for communication may achieve high concurrency but will likely lead to poor performance because of high communication costs. On the other hand, assigning all transitions to the same processor may lower communication costs but will ignore opportunities for

concurrent processing, thus failing to take advantage of the distributed architecture. Achieving high throughput will require a balance between these two extremes.

To take advantage of the processing power of the distributed architecture, each processing graph will be analyzed at compile-time. This analysis will identify connected segments in the processing graph which contain transitions that can be statically assigned and scheduled using established techniques [5, 6, 7, 8]. Directed cycles in the processing graph will also be identified. The results will be used to assign the transitions of the processing graph to the processors in the target architecture. The goal in this assignment is to provide maximum throughput within specified latency constraints. This assignment is an NP-hard problem, and so we will use heuristic methods to reach suboptimal solutions.

In analyzing and partitioning the graph at compile time, we will design PGMT to identify connected segments of transitions and connecting places for assignment to processes. This will reduce costs of both interprocess communication and run-time assignment of transitions to processors.

In most application processing graphs, we anticipate that the special transitions will be used sparingly. The reason is that most ordinary transitions will read tokens that are families containing large amounts of data, which are compatible with the output of the ordinary transitions immediately upstream. Predominant use of ordinary transitions will simplify the graph analysis. Moreover, depending on the target architecture, this may well provide an additional performance enhancement by simplifying the communication between processors.

3.1 Technical Problems

When done effectively, static analysis and assignment results in significant performance improvements by reducing the run-time overhead of assigning the processes to processors and by reducing communication costs between processors. The literature reflects much active research to find methods for such analysis and assignment, and we will take advantage of proven methods.

The compile time analysis and assignment depends on assumptions about the structure of the processing graph and the target architecture. In particular, the structure of the processing graph and the target architecture are assumed to be fixed over time.

In PGM, the varying of produce and consume amounts is limited to the special transitions. If a segment contains only ordinary transitions, which always consume and produce exactly one token at each adjacent place, then the segment returns to its previous state after each transition fires once. While partitioning of the segments and assignment to processors is not trivial, [5] provides a partial analysis of partitioning and assignment.

Analyzing the more coarse-grained structure of the segments and connections between them is more difficult, because with the special transitions, the amount of data produced and consumed varies over time. Therefore, the scheduling and assignment of these segments must be done during run time.

It is undecideable whether a dataflow graph with variable produce and consume amounts can be run forever in bounded memory. Nonetheless, Tom Parks gives a method for executing a graph in bounded memory, whenever that is possible [8].

When the command program reconfigures a processing graph, the analysis, assignment, and distribution must be repeated. If the new configuration can be anticipated at compile time, then this analysis can be accomplished beforehand. If not, then the analysis, assignment, and distribution imposes a run time cost. Compile time analysis of configurations will be limited in cases where an unmanageably large number of configurations results from many independent configuration options.

If the distributed architecture is dynamically reconfigurable (i.e., if processors may be added to or removed from the system), the assignment and distribution must be repeated. Again, a run time cost must be paid.

3.2 Status of PGMT Development Progress to Date

At NRL, we have demonstrated a GUI capture of a processing graph and execution on a Sun workstation in a multi-threaded environment. Our next step will be to target the processing to a network of Sun workstations. Following that, we plan to target a heterogeneous network of processors. Finally, we plan to target a dynamic network of processors, in which individual processors may be taken off line and others added.

4. Summary

We have described PGM as a method of specifying applications. PGM is architecture independent and iconic. It has a reconfigurable data flow model of computation. At NRL, we are building a tool set PGMT, which will demonstrate technology to implement PGM on a given distributed architecture. The cost of such implementation using PGMT will be much less than working from scratch.

We will place all of our software in the public domain, with the expectation that the commercial marketplace will build on our work. If we are successful, the power of distributed processing will be opened to a much broader audience. The entire computer industry will benefit, and the U.S. Navy will have achieved its goal of reducing the life-cycle cost of high performance software.

ACKNOWLEDGEMENTS

This work was partially supported by the Office of Naval Research Computer Science Program.

The author wishes to thank David J. Kaplan for many helpful discussions.

REFERENCES

1. D.J. Kaplan and R.S. Stevens, *Processing Graph Method 2.0 Semantics*, U.S. Naval Research Laboratory, 1995, http://www.ait.nrl.navy.mil/pgmt/PGMspectoc.html

2. Richard M. Karp & Raymond E. Miller, *Properties of a Model for Parallel Computations: Determinacy, Termination, Queuing*, SIAM J. Appl. Math. v14, pp 1390-1410, 1966.

3. D.J. Kaplan, *An Introduction to the Processing Graph Method*, 1997 (Presented at the International Conference on Engineering of Computer Based Systems, IEEE, Monterey California, March 24-28, 1997).

4. J. L. Peterson, *Petri Net Theory and the Modeling of Systems*, Englewood Cliffs, N.J., Prentice-Hall, c1981.

5. G. C. Sih, *Multiprocessor Scheduling to Account for Interprocessor Communication*, Ph.D. thesis, Memorandum No. UCB/ERL M91/29, Electronics Research Laboratory, University of California at Berkeley, April, 1991.

6. J. T. Buck, *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model*, Ph.D. thesis, Memorandum No. UCB/ERL M93/69, Electronics Research Laboratory, University of California at Berkeley, September, 1993.

7. S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, *Software Synthesis from Dataflow Graphs*, Kluwer Academic Publishers, Norwell, Ma, 1996.

8. T. M. Parks, *Bounded Scheduling of Process Networks*, Technical Report UCB/ERL-95-105. PhD Dissertation. EECS Department, University of California, Berkeley CA 94720, December 1995, http://ptolemy.eecs.berkeley.edu/papers/parksThesis.